**## Implementation Plan**

The implementation plan is organized into different phases, focusing on progressive development from core infrastructure to advanced features.

**### Phase 0 – Initial Setup and Environment Configuration**

**\*\*Objectives\*\***:

- Set up Git repository for version control and collaboration

- Set up Actinia, PostgreSQL, and Valkey services with Docker Compose

- Configure persistent storage for GRASS GIS database

- Create initial users and verify system functionality

- Prepare backend development environment

**### Prerequisites:**

- Docker and Docker Compose (latest stable versions)

docker version #Docker Desktop 4.42.0

docker --version #Docker version 28.2.2

docker compose version #v2.36.2-desktop.1

- Python 3.9+ with venv or conda

python3 –version #Python 3.13.3

- Git for version control

git –version # git version 2.45.1.windows.1

- GitHub account for collaboration

GitHub is a cloud-based platform where you **store your Git repositories online**.

- PowerShell (for Windows users)

- Basic understanding of RESTful APIs and geospatial concepts

- PostGIS/QGIS for data inspection (recommended)

**### Step 1: Git Repository Setup and Collaboration**

**\*\*1.1. Initialize Local Git Repository\*\***:

```powershell

# Navigate to your project directory

cd c:\Users\Lorenzo\Documents\python\_envs\rest\_grass\_gis

cd D:\thesis\rest\_grass\_gis

# Initialize git repository

git init

# Configure git user (if not already done globally)

git config user.name "Your Name"

git config user.email [your.email@example.com](mailto:your.email@example.com)

git config user.name "Farinaz Goli"

git config user.email "ffarinazgolii@gmail.com"

```

**\*\*1.2. Create Initial .gitignore\*\***:

The .gitignore file tells Git which files and folders to ignore (not track).

This is useful for keeping log files, temporary files, build artifacts, or personal files out of your repository. When to Use .gitignore

* When you want to keep sensitive, local, or unnecessary files out of your repository
* When sharing a project with others and want to avoid cluttering Git history
* When working with build tools or editors that create extra files

Create `.gitignore` file:

```gitignore

# Python

\_\_pycache\_\_/

\*.py[cod]

\*$py.class

\*.so

.Python

env/

venv/

ENV/

env.bak/

venv.bak/

# Environment variables

.env

.env.local

.env.\*.local

# Database

\*.db

\*.sqlite

# Docker volumes and data

data/

!data/.gitkeep

# Uploads and temporary files

uploads/

temp/

\*.tmp

# Logs

\*.log

logs/

# OS generated files

.DS\_Store

.DS\_Store?

.\_\*

.Spotlight-V100

.Trashes

ehthumbs.db

Thumbs.db

# IDE

.vscode/

.idea/

\*.swp

\*.swo

# Testing

.pytest\_cache/

.coverage

htmlcov/

.tox/

# Actinia cloned repository (temporary)

actinia-core/

# Docker

.docker/

```

**\*\*1.3. Create GitHub Repository and Connect\*\***:

1. Go to GitHub.com and create new repository: `rest-grass-gis-solar-api`

2. Set as **\*\*Private\*\*** repository

3. Description: `FastAPI service for solar radiation analysis using GRASS GIS via Actinia`

4. Do NOT initialize with README

```powershell

# Add GitHub remote (replace YOUR\_USERNAME with your GitHub username)

git remote add origin https://github.com/YOUR\_USERNAME/rest-grass-gis-solar-api.git

git remote add origin <https://github.com/Farinaz94/rest-grass-gis-solar-api.git>

Hey Git, please **link this local project** to a **GitHub repository** at this address, and I’ll call it origin.”

Once this is set, you can:

* **Push** your local code to GitHub
* **Pull** updates from GitHub if needed

# Verify remote was added

git remote -v

output:  
origin https://github.com/Farinaz94/rest-grass-gis-solar-api.git (fetch)

origin https://github.com/Farinaz94/rest-grass-gis-solar-api.git (push)

```

**\*\*1.4. Add Collaborator Access\*\***:

1. Go to your repository settings on GitHub

2. Click "Collaborators" → "Add people"

3. Add instructor's GitHub username

4. Send invitation

**### Step 2: Actinia Docker Image Preparation**

**\*\*2.1. Clone Actinia Repository\*\***:

```powershell

# Clone actinia-core repository

git clone https://github.com/actinia-org/actinia-core.git

cd actinia-core

```

**\*\*2.2. Modify Dockerfile for Solar Extensions\*\***:

Editing the Dockerfile used to build the Actinia image. It means Install a GRASS GIS module called r.sun.hourly to calculate **hourly solar radiation**

Docker images are read-only templates that contain instructions for creating a container. A Docker image is a snapshot or blueprint of the libraries and dependencies required inside a container for an application to run.

Edit `.\docker\actinia-core-alpine\Dockerfile` and add the following lines at line 24:

```dockerfile

RUN grass --tmp-location EPSG:4326 --exec g.extension -s \

    extension=r.sun.hourly

RUN grass --tmp-location EPSG:4326 --exec g.extension extension=r.sun.hourly

```

**\*\*2.3. Build Custom Actinia Image\*\***:

```powershell

docker build -f .\docker\actinia-core-alpine\Dockerfile --tag actinia:local .

cd ..

```

Ran, but 8 warnings:  
“8 warnings found (use docker --debug to expand):

- FromAsCasing: 'as' and 'FROM' keywords' casing do not match (line 43)

- LegacyKeyValueFormat: "ENV key=value" should be used instead of legacy "ENV key value" format (line 9)

- LegacyKeyValueFormat: "ENV key=value" should be used instead of legacy "ENV key value" format (line 12)

- LegacyKeyValueFormat: "ENV key=value" should be used instead of legacy "ENV key value" format (line 13)

- FromAsCasing: 'as' and 'FROM' keywords' casing do not match (line 1)

- FromAsCasing: 'as' and 'FROM' keywords' casing do not match (line 2)

- FromAsCasing: 'as' and 'FROM' keywords' casing do not match (line 4)

- FromAsCasing: 'as' and 'FROM' keywords' casing do not match (line 32)”

**### Step 3: Project Structure and Data Directory Setup**

**\*\*3.1. Create Project Directory Structure\*\***:

```powershell

# Create data directory with .gitkeep to track empty folders

cd D:\thesis\rest\_grass\_gis

mkdir data

New-Item -ItemType File -Path data\.gitkeep

# Create main data directory with subdirectories

mkdir -p data\actinia-data\grassdb

mkdir -p data\actinia-data\resources

mkdir -p data\actinia-data\userdata

mkdir -p data\actinia-data\workspace\temp\_db

mkdir -p data\actinia-data\workspace\tmp

mkdir -p data\valkey\_data\config

# Create backend directory structure placeholder

mkdir backend

New-Item -ItemType File -Path backend\.gitkeep

# Create documentation directory

mkdir docs

New-Item -ItemType File -Path docs\.gitkeep

```

**\*\*3.2. Copy Valkey Configuration from Actinia Repository\*\***:

Valkey is:

* An **open-source, in-memory data store** that supports various data structures: strings, hashes, lists, sets, etc.
* Designed for **speed**, with **sub-millisecond latency** for reading/writing data.

**In Actinia:**

In your Actinia setup, Valkey acts like a **temporary brain**:

* It stores **user session data**, **task statuses**, and **temporary GIS processing results**.
* Without Valkey, Actinia can’t efficiently track user interactions or background job progress.

```powershell

# Copy the entire valkey\_data folder from actinia repository

cd D:\thesis\rest\_grass\_gis

Copy-Item -Path .\actinia-core\docker\valkey\_data -Destination .\data\ -Recurse

**Source**: .\actinia-core\docker\valkey\_data

**Destination**: .\data\ #the folder that we made

-Recurse: this ensures all subfolders and files are copied too.

```

**\*\*3.3. Fix Valkey Password File\*\***:

```powershell

# The copied password file may contain extra lines that cause authentication errors

# Edit the .valkey file to contain only the password

notepad .\data\valkey\_data\config\.valkey

#basd on <https://github.com/actinia-org/actinia-docker/blob/main/valkey_data/config/.valkey>, it should be pass

# Ensure the file contains ONLY the password, no extra lines or characters

# Save and close the file

```

**\*\*3.4. Initial Git Commit\*\***:

**What is a Git Commit?**

A **commit** in Git is like **taking a snapshot** of your project at a specific point in time.

When you commit:

* You **save changes** you've made to files (after staging them).
* Git **records** those changes in the repository history.
* Each commit has a **message** that explains *what* you changed and *why*.
* A **Git commit** is:
* A save point in your project,
* With a message explaining what you did,
* So you can track and manage changes over time.

```powershell

# Add all files to staging

cd D:\thesis\rest\_grass\_gis

what is “implementation\_phases.md”? inside this file should be empty?

This command **adds specific files to Git's staging area** — the place where you prepare files before committing them into the Git history.

git add .gitignore implementation\_phases.md data\.gitkeep backend\.gitkeep docs\.gitkeep

ERROR! fatal: pathspec 'implementation\_phases.md' did not match any files

Solution:  
notepad implementation\_phases.md #wrote # Implementation Phases inside the notepad

And then save.

ERROR! fatal: pathspec 'data\.gitkeep' did not match any files

Solution:

New-Item -ItemType File -Path .\data\.gitkeep

# Create initial commit

git commit -m "Initial project setup with implementation phases documentation and directory structure"

# Set default branch name to main

git branch -M main

# Push to GitHub

git push -u origin main

```

**### Step 4: Docker Compose Configuration**

**\*\*4.1. Create docker-compose.yml\*\***:

this file tells Docker how to **run multiple connected services** (like Actinia, Redis/Valkey, and PostgreSQL) **together**. this file will let you **launch all the services Actinia needs** with one command:

* actinia → the main Actinia container, using your custom image
* valkey → Redis-style key-value store (for caching or temp data)
* postgres → PostgreSQL with GIS (PostGIS) support, for storing spatial data
* volumes: at the end make sure data is **persisted** (doesn’t get deleted every time the container stops).

```yaml

services:

  actinia:

    image: actinia:local

    ports:

      - "8088:8088"

    environment:

      - ACTINIA\_CUSTOM\_TEST\_SERVER=true

      - ACTINIA\_API\_LOG\_LEVEL=debug

    volumes:

      - ./data/actinia-data/grassdb:/actinia\_core/grassdb

      - ./data/actinia-data/resources:/actinia\_core/resources

      - ./data/actinia-data/userdata:/actinia\_core/userdata

      - ./data/actinia-data/workspace/temp\_db:/actinia\_core/workspace/temp\_db

      - ./data/actinia-data/workspace/tmp:/actinia\_core/workspace/tmp

    restart: unless-stopped

    depends\_on:

      valkey:

        condition: service\_started

      postgres:

        condition: service\_healthy

    healthcheck:

      test: ["CMD", "curl", "-f", "http://localhost:8088/api/v3/version"]

      interval: 30s

      timeout: 10s

      retries: 5

      start\_period: 60s

  valkey:

    image: valkey/valkey:8.1-alpine

    volumes:

      - ./data/valkey\_data:/data

    environment:

      - VALKEY\_PASS\_FILE=/data/config/.valkey

    command: [

      "sh", "-c",

      "docker-entrypoint.sh /data/config/valkey.conf --requirepass \"$$(cat $$VALKEY\_PASS\_FILE)\""

    ]

    restart: unless-stopped

    healthcheck:

      test: ["CMD", "valkey-cli", "ping"]

      interval: 10s

      timeout: 5s

      retries: 3

  postgres:

    image: postgis/postgis:14-3.2

    ports:

      - "5432:5432"

    environment:

      - POSTGRES\_USER=actinia

      - POSTGRES\_PASSWORD=actinia

      - POSTGRES\_DB=actinia

    volumes:

      - postgres\_data:/var/lib/postgresql/data

    restart: unless-stopped

    healthcheck:

      test: ["CMD-SHELL", "pg\_isready -U actinia"]

      interval: 10s

      timeout: 5s

      retries: 5

volumes:

  postgres\_data:

```

**\*\*4.2. Create Environment Configuration\*\***:

Create `.env` file:

```env

# Actinia Configuration

ACTINIA\_URL=http://localhost:8088

ACTINIA\_USER=actinia-gdi

ACTINIA\_PASSWORD=actinia-gdi

# Database Configuration

POSTGRES\_HOST=localhost

POSTGRES\_PORT=5432

POSTGRES\_USER=actinia

POSTGRES\_PASSWORD=actinia

POSTGRES\_DB=actinia

DATABASE\_URL=postgresql://actinia:actinia@localhost:5432/actinia

# Valkey Configuration

VALKEY\_HOST=localhost

VALKEY\_PORT=6379

VALKEY\_PASSWORD=your\_secure\_valkey\_password\_here

# Development Settings

DEBUG=true

LOG\_LEVEL=INFO

```

**### Step 5: Launch and Verify Services**

**\*\*5.1. Start Docker Services\*\***:

```powershell

# Start all services

docker-compose up -d

# Monitor startup logs

docker-compose logs -f

``

**\*\*5.2. Service Health Verification\*\***:

WSL2:

# 1) Check that all Compose services are up

# (acts like `docker ps` but filtered through your compose file)

docker compose ps

# 2) Verify the Actinia API is alive

curl http://localhost:8088/api/v3/version

# 3) Test the PostgreSQL/PostGIS connection

# exec into the 'postgres' container and run psql

docker compose exec postgres \

psql -U actinia -d actinia -c "SELECT version();"

# 4) Test the Valkey (Redis) connection

# exec into the 'valkey' container and ping it

docker compose exec valkey \

valkey-cli ping

```powershell

# Check all containers are running

docker-compose ps

# Verify Actinia API is responding

curl http://localhost:8088/api/v3/version

# Alternative for PowerShell users

Invoke-WebRequest -Uri "http://localhost:8088/api/v3/version"

# Test PostgreSQL connection

docker-compose exec postgres psql -U actinia -d actinia -c "SELECT version();"

# Test Valkey connection

docker-compose exec valkey valkey-cli ping

(error) NOAUTH Authentication required.

```

**### Step 6: User Management Setup**

**\*\*6.1. Create Initial Actinia Users\*\***:

# 1) Create the admin user

docker compose exec actinia \

actinia-user create \

-u admin \

-w admin123 \

-r admin \

-g admin \

-c 1000000000 \

-n 10000 \

-t 60000

# 2) Create the development user

docker compose exec actinia \

actinia-user create \

-u dev\_user \

-w dev123 \

-r user \

-g solar\_team \

-c 100000000 \

-n 1000 \

-t 6000

# 3) Create the test user for your API tests

docker compose exec actinia \

actinia-user create \

-u test\_user \

-w test123 \

-r user \

-g solar\_team \

-c 100000000 \

-n 1000 \

-t 6000

For listing all the users:  
docker compose exec actinia actinia-user list

```powershell

# Create admin user

docker-compose exec actinia actinia-user create -u admin -w admin123 -r admin -g admin -c 1000000000 -n 10000 -t 60000

# Create development user

docker-compose exec actinia actinia-user create -u dev\_user -w dev123 -r user -g solar\_team -c 100000000 -n 1000 -t 6000

# Create test user for API testing

docker-compose exec actinia actinia-user create -u test\_user -w test123 -r user -g solar\_team -c 100000000 -n 1000 -t 6000

```

**\*\*User Parameters Explanation:\*\***

- `-u`: Username

- `-w`: Password

- `-r`: Role (user, admin, superadmin)

- `-g`: Group name

- `-c`: Cell limit (max raster cells per process)

- `-n`: Process number limit (max concurrent processes)

- `-t`: Time limit per process (seconds)

**\*\*6.2. Verify User Authentication\*\***:

```powershell

# Test user login via Actinia API

curl -X POST "http://localhost:8088/api/v3/auth" -H "Content-Type: application/json" -d '{"username": "dev\_user", "password": "dev123"}'

# This should return an authentication token

```

**### Step 7: Backend Development Environment Setup**

**\*\*7.1. Create Backend Directory Structure\*\***:

```powershell

# Create backend directory

mkdir backend #I had it before

cd backend

# Create Python virtual environment

python -m venv venv

# Activate virtual environment

.\venv\Scripts\Activate.ps1

ERROR! running scripts is disabled on this system

Solution:

1. Open PowerShell **as Administrator** (important!)
2. Run this command: Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
3. When prompted, type Y and press Enter.
4. Then close and re-open PowerShell (not admin this time), go to your project folder:

cd D:\thesis\rest\_grass\_gis\backend

.\venv\Scripts\Activate.ps1

# Create basic project structure

mkdir -p app\api\v1, app\core, app\models, app\schemas, app\services

mkdir -p tests, config, docs

# Create \_\_init\_\_.py files

New-Item -ItemType File -Path app\\_\_init\_\_.py

New-Item -ItemType File -Path app\api\\_\_init\_\_.py

New-Item -ItemType File -Path app\api\v1\\_\_init\_\_.py

```

**\*\*7.2. Install Development Dependencies\*\***:

Create `requirements.txt`:

New-Item -ItemType File -Path requirements.txt

```txt

fastapi==0.104.1

uvicorn[standard]==0.24.0

pydantic==2.5.0

pydantic-settings==2.1.0

sqlalchemy==2.0.23

alembic==1.13.0

psycopg2-binary==2.9.9

httpx==0.25.2

python-dotenv==1.0.0

pytest==7.4.3

pytest-asyncio==0.21.1

python-jose[cryptography]==3.3.0

passlib[bcrypt]==1.7.4

python-multipart==0.0.6

```

Then install all dependencies:

pip install -r requirements.txt

ERROR! pg\_config executable not found.

Solution:

It was because of python version 13.x.x. I downgraded the python to 11.x.x and then it works

**### Step 8: Understanding Actinia Core Concepts**

**\*\*8.1. Explore Permanent vs Temporary GRASS Databases\*\***:

WSL2

For checking existing projects.

curl -u admin:admin123 \

http://localhost:8088/api/v3/locations

# 1) Create a new GRASS project (location) named “test-proj” in EPSG:4326

curl -u admin:admin123 -X POST \

-H "Content-Type: application/json" \

-d '{"location\_name":"test-proj","epsg":"4326"}' \

http://localhost:8088/api/v3/locations/test-proj

# ↳ Response: confirms creation (or error if it already exists)

# 2) Provision the mandatory “PERMANENT” mapset inside that location

curl -u admin:admin123 -X POST \

http://localhost:8088/api/v3/locations/test-proj/mapsets/PERMANENT

# ↳ Response: “Mapset <PERMANENT> exists.” if it was auto-created, or success if new.

# 3) Create your user/temporary mapset called “test\_map”

curl -u admin:admin123 -X POST \

http://localhost:8088/api/v3/locations/test-proj/mapsets/test\_map

# ↳ Response: “Mapset <test\_map> successfully created.”

# 4) Verify both mapsets now exist

curl -u admin:admin123 \

http://localhost:8088/api/v3/locations/test-proj/mapsets

# ↳ Should return: ["PERMANENT","test\_map"]

```powershell

# List permanent locations

curl -H "Authorization: Bearer YOUR\_TOKEN" http://localhost:8088/api/v3/locations

# Create a temporary location for testing

curl -X POST -H "Authorization: Bearer YOUR\_TOKEN" -H "Content-Type: application/json" \

  -d '{"location\_name":"temp\_location","epsg":"4326"}' \

  http://localhost:8088/api/v3/locations/temp\_location

# Understand mapset structure

curl -H "Authorization: Bearer YOUR\_TOKEN" http://localhost:8088/api/v3/locations/temp\_location/mapsets

```

**\*\*8.2. Test User Role Permissions\*\***:

WSL2:

For admin:  
curl -u dev\_user:dev123 \

<http://localhost:8088/api/v3/locations>

For admin:

url -u admin:admin123 \

http://localhost:8088/api/v3/locations

```powershell

# Test as regular user (should have limited access)

curl -X POST -H "Content-Type: application/json" \

  -d '{"username": "dev\_user", "password": "dev123"}' \

  http://localhost:8088/api/v3/auth

# Test as admin user (should have full access)

curl -X POST -H "Content-Type: application/json" \

  -d '{"username": "admin", "password": "admin123"}' \

  http://localhost:8088/api/v3/auth

```

**\*\*Expected Project Structure After Phase 0:\*\***

```

rest\_grass\_gis\

├── .git/                       # Git repository

├── .gitignore                  # Git ignore rules

├── actinia-core/               # Cloned repository (ignored by git)

├── data/                       # Docker persistent data (ignored by git)

│   ├── .gitkeep               # Track empty directory

│   ├── actinia-data/

│   │   ├── grassdb/           # Persistent GRASS database

│   │   ├── resources/

│   │   ├── userdata/

│   │   └── workspace/

│   └── valkey\_data/

│       └── config/

│           ├── valkey.conf

│           └── .valkey        # Password file

├── backend/                    # FastAPI application

│   ├── .gitkeep               # Track empty directory initially

│   ├── venv/                  # Python virtual environment (ignored by git)

│   ├── app/

│   ├── tests/

│   └── requirements.txt

├── docs/                       # Project documentation

│   └── .gitkeep

├── docker-compose.yml

├── .env                        # Environment variables (ignored by git)

├── README.md                   # Project overview

└── implementation\_phases.md    # This file

```

**### Deliverables for Phase 0:**

-  Git repository initialized and connected to GitHub

-  Collaborator access granted to instructor

-  Project directory structure created and committed

-  Actinia service with r.sun.hourly extension running on port 8088

-  PostgreSQL with PostGIS running on port 5432

-  Valkey service for session management

-  Three test users created (admin, dev\_user, test\_user)

-  Persistent GRASS database storage configured

-  Backend development environment prepared

-  All services verified and responding to health checks

-  Understanding of permanent vs temporary GRASS databases in Actinia

-  Understanding of user roles and permissions in Actinia

**### Phase 1 – FastAPI Setup and Hybrid Authentication System**

**JWT** stands for **JSON Web Token**. It’s a secure way to **authenticate users** and manage **sessions** in APIs like FastAPI.

**\*\*Objectives\*\***:

- Create FastAPI application structure with proper configuration

- Implement hybrid authentication system: validate with Actinia, issue JWT tokens

- Design and implement database schema for user profiles and job tracking

- Set up development workflow with hot reload and testing

**\*\*Key Tasks\*\***:

- Create FastAPI application with proper project structure

- Implement two-step authentication: Actinia validation + JWT token generation

- Design database models for user profiles, sessions, jobs, and metadata

- Set up database migrations with Alembic

- Create authentication endpoints and user profile management

- Implement development workflow with auto-reload

**\*\*Implementation Details\*\***:

1. **\*\*Hybrid Authentication Architecture\*\***:

   Implement authentication system where FastAPI validates credentials against Actinia, then generates internal JWT tokens with extended claims for session management and user experience features.

2. **\*\*Database Schema for User Management\*\***:

   Design SQLAlchemy models for user profiles (linked to Actinia usernames), session tracking, job ownership, and extended metadata. Store user preferences and profile data while keeping credentials in Actinia.

3. **\*\*JWT Token Management\*\***:

   Create JWT token generation, validation, and refresh mechanisms with configurable expiration policies and secure session management.

**\*\*Step 1: FastAPI Application Setup\*\***

**\*\*1.1. Create FastAPI Application Structure\*\***:

WSL2

cd backend

mkdir -p app

touch app/main.py app/config.py app/database.py

mkdir -p app/api/auth

mkdir -p app/api/v1/users

mkdir -p app/api/v1/jobs

mkdir -p app/api/v1/health

touch app/api/auth/\_\_init\_\_.py

touch app/api/v1/users/\_\_init\_\_.py

touch app/api/v1/jobs/\_\_init\_\_.py

touch app/api/v1/health/\_\_init\_\_.py

mkdir -p app/services/actinia app/services/auth app/services/database

touch app/services/actinia/\_\_init\_\_.py

touch app/services/auth/\_\_init\_\_.py

touch app/services/database/\_\_init\_\_.py

mkdir -p app/models app/schemas

touch app/models/\_\_init\_\_.py

touch app/schemas/\_\_init\_\_.py

```powershell

cd backend

# Create core application files

New-Item -ItemType File -Path app\main.py

New-Item -ItemType File -Path app\config.py

New-Item -ItemType File -Path app\database.py

# Create API structure for hybrid auth

mkdir app\api\auth, app\api\v1\users, app\api\v1\jobs, app\api\v1\health

New-Item -ItemType File -Path app\api\auth\\_\_init\_\_.py

New-Item -ItemType File -Path app\api\v1\users\\_\_init\_\_.py

New-Item -ItemType File -Path app\api\v1\jobs\\_\_init\_\_.py

New-Item -ItemType File -Path app\api\v1\health\\_\_init\_\_.py

# Create service layer with auth services

mkdir app\services\actinia, app\services\auth, app\services\database

New-Item -ItemType File -Path app\services\actinia\\_\_init\_\_.py

New-Item -ItemType File -Path app\services\auth\\_\_init\_\_.py

New-Item -ItemType File -Path app\services\database\\_\_init\_\_.py

# Create models and schemas

New-Item -ItemType File -Path app\models\\_\_init\_\_.py

New-Item -ItemType File -Path app\schemas\\_\_init\_\_.py

```

**\*\*1.2. Update Requirements for JWT Support\*\***:

Add to `requirements.txt`:

```txt

python-jose[cryptography]==3.3.0

passlib[bcrypt]==1.7.4

python-multipart==0.0.6

```

**\*\*Step 2: Hybrid Authentication Implementation\*\***

**\*\*2.1. Implement Actinia Validation Service\*\***:

Create service for validating credentials against Actinia and retrieving user information and permissions.

**\*\*2.2. Create JWT Token Service\*\***:

Develop JWT token generation, validation, and refresh functionality with configurable claims and expiration.

**\*\*2.3. Implement Authentication Endpoints\*\***:

Create `/auth/login`, `/auth/refresh`, `/auth/logout`, and `/auth/me` endpoints for complete authentication flow.

**\*\*Step 3: Enhanced Database Schema\*\***

**\*\*3.1. Create User Management Models\*\***:

Design SQLAlchemy models for:

- Users (profile data linked to Actinia username, preferences, settings)

- Sessions (JWT token metadata, device tracking, login history)

- Jobs (enhanced with user relationships instead of string usernames)

- Files (user ownership and access control)

- Results (user-specific result storage and sharing permissions)

**\*\*3.2. Setup Database Migrations\*\***:

Initialize Alembic with enhanced schema supporting user profiles and session management.

**\*\*3.3. User Profile Synchronization\*\***:

Implement functions to sync user profile data with Actinia user information and handle role mapping.

What does Step 3.3 mean?

You already have a User table in your PostgreSQL database. Now we want to fill it with real users — specifically, those who log in through Actinia.

Think of it like this:

* Actinia knows who the users are (with their usernames and roles).
* But your local database still doesn’t know anything about them.
* So in this step, we will copy the user info from Actinia into your own database.

What do we need to code?

You’ll write a **small Python function** that does this:

def sync\_user\_from\_actinia(actinia\_token: str):

# 1. Call Actinia API to get user info

# 2. Check if user exists in DB

# 3. If not, insert them

# 4. If yes, maybe update their info

**\*\*Step 4: User Management Features\*\***

**\*\*4.1. User Profile Management\*\***:

Create endpoints for user profile CRUD operations, preferences management, and account settings.

**\*\*4.2. Session Management\*\***:

Implement session tracking, device management, and security features like session invalidation.

**\*\*4.3. Enhanced Authorization\*\***:

Develop role-based access control and permission management building on Actinia's user roles.

**\*\*Deliverables\*\***:

- FastAPI application with hybrid authentication system

- JWT token-based authentication with Actinia credential validation

- Enhanced database schema with user profiles and session management

- User management endpoints and profile functionality

- Development environment with comprehensive authentication testing

- Session security features and token refresh mechanisms

**### Phase 2 – File Upload, Validation Pipeline, and Synthetic DSM Generation**

**\*\*Objectives\*\***:

- Implement file upload endpoints with validation using GDAL

- Create GRASS location/mapset management through Actinia

- Develop raster and vector data import functionality

- Implement synthetic DSM generation from building footprints when no DSM is available

- Implement basic terrain analysis capabilities (slope/aspect)

**\*\*Key Tasks\*\***:

- Create file upload endpoint with multi-format support (GeoTIFF, Shapefile, GeoJSON)

- Implement GDAL-based validation for spatial data integrity and projection

- Develop location/mapset creation and management through Actinia API

- Create import process chains for raster and vector data

- Implement synthetic DSM creation workflow for buildings without elevation data

- Extract building heights from existing DSM when available

- Implement slope and aspect calculation workflows for both synthetic and real DSM

- Design file metadata storage and retrieval system

**\*\*Implementation Details\*\***:

1. **\*\*File Upload and Validation System\*\***:

   Develop upload endpoints that accept multiple geospatial formats, validate them using GDAL for proper structure and projection information, and store file metadata in the database with validation results.

2. **\*\*GRASS Location Management\*\***:

   Create functions to manage GRASS locations and mapsets via Actinia, including automatic location creation based on uploaded data projection and mapset organization per user session.

3. **\*\*Import Process Chains\*\***:

   Design Actinia process chains for importing validated files into GRASS database, handling both raster and vector data with appropriate region settings and coordinate system management.

4. **\*\*Synthetic DSM Creation Workflow\*\***:

   Implement comprehensive synthetic DSM generation when no 5m DSM is available:

**\*\*Step 1: Height Value Assignment\*\***

   - Extract height values from building vector data attributes

   - Generate random slope values for each building within specified boundaries (e.g., 23° < slope < 30°)

**\*\*Step 2: Progressive Buffer Generation\*\***

   - Create reducing buffers of 0.5m intervals from building footprints until no new buffer can be created

   - Calculate height values for each buffer based on slope angle and distance from building edge

   - Assign progressive height increases from footprint to outer buffers

**\*\*Step 3: Rasterization and Aggregation\*\***

   - Rasterize all buffers with their respective height values

   - Use r.series to sum all buffer layers from footprints to outer buffers

   - Create composite synthetic DSM from aggregated buffer heights

**\*\*Step 4: Surface Smoothing\*\***

   - Apply r.neighbors for surface smoothing to create realistic elevation transitions

   - Configure smoothing parameters to maintain building characteristics while removing artifacts

1. **\*\*Height Extraction from Existing DSM\*\***:

When 5m DSM is available, extract mean height values within building footprint polygons using zonal statistics and assign to building attributes.

6. **\*\*Terrain Analysis for Both DSM Types\*\***:

   Implement slope and aspect calculation workflows that work with both synthetic DSM and provided DSM data, with appropriate parameter adjustments for each data type.

**\*\*Deliverables\*\***:

- Multi-format file upload endpoints with GDAL validation

- Automatic location/mapset creation based on data projection

- Raster and vector import workflows through Actinia

- Complete synthetic DSM generation pipeline with progressive buffering and height assignment

- Height extraction workflow for existing DSM data

- Terrain analysis endpoints (slope/aspect) for both synthetic and real DSM

- File metadata tracking and job status monitoring

- Error handling for invalid or corrupted spatial data